A little bit of Lisp

References

- Lisp, 3rd edition, Winston and Horn (LWH)
- Common Lisp, The Language, 2nd edition, Guy L. Steele (online)
- Beating the Averages, Paul Graham

Berthe Y. Choueiry (Shu-we-ri)
CSE, UNL

Features of Lisp

1. Interactive: interpreted and compiled
2. Symbolic
3. Functional
4. Second oldest language but still 'widely' used
   (Emacs, AutoCad, MacSyma, Yahoo Store, Orbitz, etc.)

Software/Hardware

- We have Allegro Common Lisp (by Franc Inc.): alisp and mlisp
- There are many old and new dialects (CormanLisp, Kyoto CL, LeLisp, CMU CL, SBCL, ECL, OpenMCL, CLISP, etc.)
- There have also been Lisp machines (Symbolics, Connection Machine, IT Explorer, others?)

Lisp as a functional language

`(function-name arg1 arg2` *`etc`*`)`

1. Evaluate arguments
2. evaluate function with arguments
3. return the result

Functions as arguments to other functions:

`(name2 (name1 arg1 arg2` *`etc`*`) arg3 arg2` *`etc`*`)`

Symbols

A symbol is a data structure with:

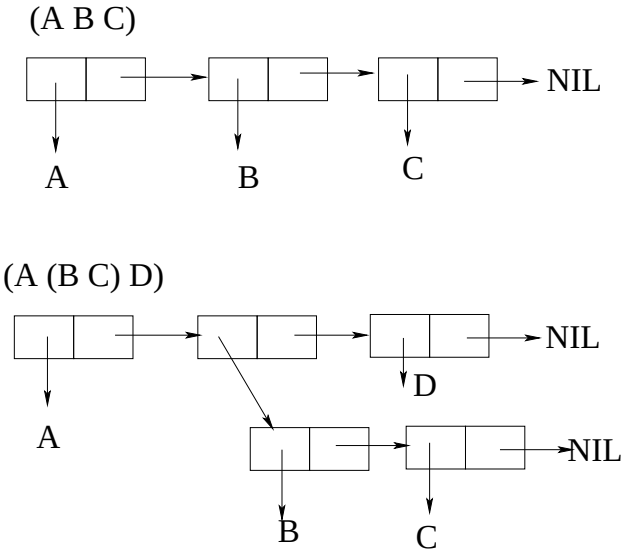
- a name
- a value
- a function
- a package, and
- a property list (plist)

```
(setf mysymbol 'hello-world)                HELLO-WORLD
(defun mysymbol (x) (* x x))                MYSYMBOL
(symbol-name 'mysymbol)                     "MYSYMBOL"
(symbol-value 'mysymbol)                    HELLO-WORLD
(symbol-function 'mysymbol)                 #<Interpreted Function MYSYMBOL>
(symbol-package 'mysymbol)                  #<The COMMON-LISP-USER package>
(funcall (symbol-function 'mysymbol) 5)     25
(find-symbol "HELLO-WORLD")                 HELLO-WORLD
                                            :INTERNAL
```

Symbolic language

- Atoms: numeric atoms (numbers), symbolic atoms (symbols)

- Lists:



Symbolic expressions: symbols and lists

More constructs

- Data types:
  atoms and lists, packages, strings, structures, vectors,
  bit-vectors, arrays, streams, hash-tables, classes (CLOS), etc.
  NIL, T, numbers, strings: special symbols, evaluate to self
- Basic functions:
  `first (car), rest (cdr), second, tenth`
  `setf`: does not evaluate first argument
  `cons, append, equal`, operations on sets, *etc.*
- Basic macros:
  `defun, defmacro, defstruct, defclass, defmethod,`
  `defvar, defparameter`

Special forms: `let`, `let*`, `flet`, `labels`, `progn`,

```
(defun mytest (item)
  (let* ((item '(1 2 3 4))
         (item2 (append item item)))
    (format t "~%Item is ~a" item)
    (format t "~%Item2 is ~a" item2))
  (format t "~%Item is ~a" item))

(mytest 'mysymbol)

Item is (1 2 3 4)
Item2 is (1 2 3 4 1 2 3 4)
Item is MYSYMBOL
NIL
```

- Predicates:
  `listp`, `endp`, `atom`, `numberp`, `symbolp`, `evenp`, `oddp`, *etc.*
- Conditionals:
  `if <test> <then form> <else form>`,
  `when <test> <then form>`,
  `unless <test> <else form>`,
  `cond`,
  `case`

```
(if (oddp 3) (* 3 2) 3)
6

(cond ((listp mysymbol) (format t "It is a list"))
      ((numberp mysymbol) (format t "It is a number"))
      ((atom mysymbol) (format t "It is an atom"))
      (t (format "It is something else")))
It is an atom
NIL
```

Looping constructs: `dolist`, `dotimes`, `do`, `mapcar`, `loop`

```
(let ((list '(1 2 3 4 5))
      (result nil))
  (dolist (item list (reverse result))
    (push (* item item) result)))

(let ((list '(1 2 3 4 5))
      (result nil))
  (do ((mylist list (rest mylist)))
      ((endp mylist) (reverse result))
    (push (* (first mylist) (first mylist)) result)))

(mapcar #'(lambda(x) (* x x)) '(1 2 3 4 5))
```

Lambda functions!!

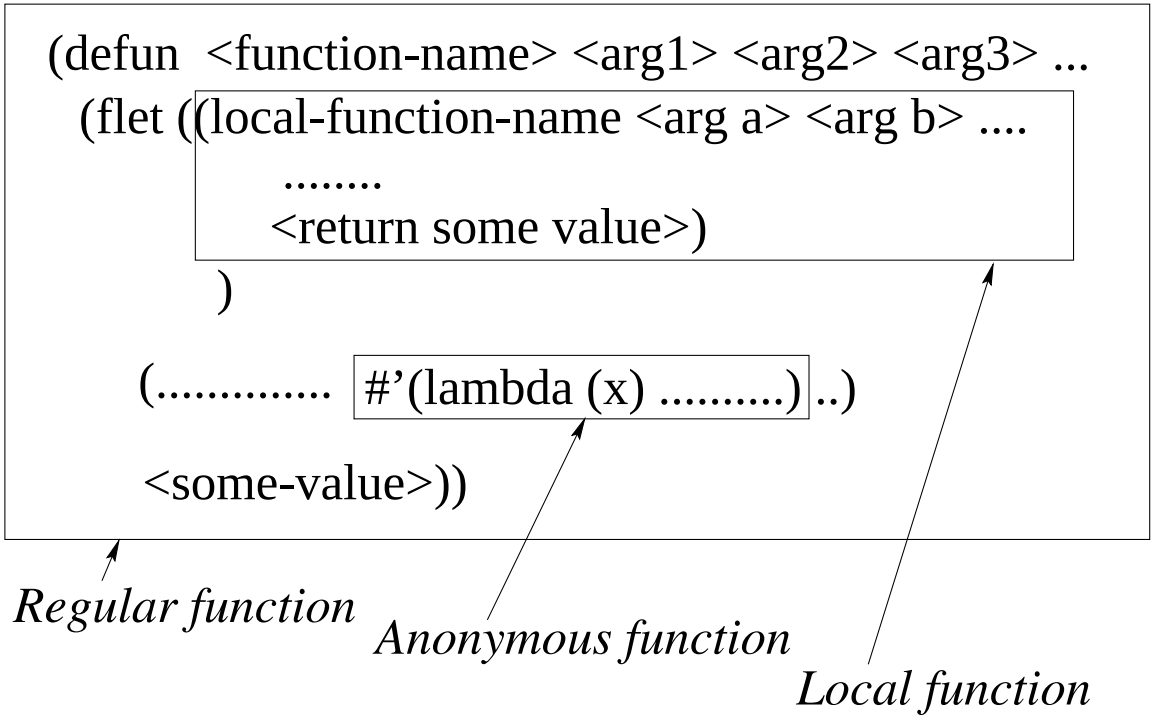
My favorite functions

- mapcar: unary function and alist, generates a new list obtained by applying the function to each element in the list
- reduce: binary function and alist, applies the function to the first two elements of the list, then to the result and the third element, then to the result and the fourth element, etc.

```
(mapcar #'(lambda(x) (* x x)) '(1 2 3 4 5))

(reduce #'(lambda(x y) (if (> x y) x y)) '(1 5 2 6 3 7 4))
```

A really functional language



`defun, flet/labels, lambda`

What makes Lisp different?

Paradigms of AI Programming, Norvig

- Built-in support for lists
- Dynamic storage management (garbage collection!)
- Dynamic typing
- First-class functions (dynamically created, anonymous)
- Uniform syntax
- Interactive environment
- Extensibility

Allegro Common Lisp

- Free download: `www.franz.com/downloads/`
- Available on Linux (cse.unl.edu).
- Great integration with emacs
  Check `www.franz.com/emacs/` Check commands distributed by instructor

- Great development environment
  Composer: debugger, inspector, time/space profiler, etc.
  `(require 'composer)`

```
;;; -*- Package: USER; Mode: LISP; Base: 10; Syntax: Common-Lisp -*-


(in-package "USER")


;;;; +=================================================+
;;;; | Source code for the farmer, wolf, goat, cabbage problem |
;;;; |     from Luger's "Artificial Intelligence, 4th Ed."     |
;;;; |  In order to execute, run the function CROSS-THE-RIVER  |
;;;; +=================================================+
```

```
;;; +=============================================+
;;; | State definitions and associated predicates |
;;; +=============================================+
(defun make-state (f w g c)
  (list f w g c))

(defun farmer-side (state)
  (nth 0 state))

(defun wolf-side (state)
  (nth 1 state))

(defun goat-side (state)
  (nth 2 state))

(defun cabbage-side (state)
  (nth 3 state))
```

```
;;; +======================+
;;; | Operator definitions |
;;; +======================+

(defun farmer-takes-self (state)
  (safe (make-state (opposite (farmer-side state))
    (wolf-side state)
    (goat-side state)
    (cabbage-side state))))

(defun farmer-takes-wolf (state)
  (cond ((equal (farmer-side state) (wolf-side state))
         (safe (make-state (opposite (farmer-side state))
                           (opposite (wolf-side state))
                           (goat-side state)
                           (cabbage-side state))))
         (t nil)))
```

```
(defun farmer-takes-goat (state)
  (cond ((equal (farmer-side state) (goat-side state))
         (safe (make-state (opposite (farmer-side state))
                           (wolf-side state)
                           (opposite (goat-side state))
                           (cabbage-side state))))
        (t nil)))

(defun farmer-takes-cabbage (state)
  (cond ((equal (farmer-side state) (cabbage-side state))
         (safe (make-state (opposite (farmer-side state))
                           (wolf-side state)
                           (goat-side state)
                           (opposite (cabbage-side state)))))
        (t nil)))
```

```
;;; +====================+
;;; | Utility functions |
;;; +====================+

(defun opposite (side)
  (cond ((equal side 'e) 'w)
        ((equal side 'w) 'e)))

(defun safe (state)
  (cond ((and (equal (goat-side state) (wolf-side state))
              (not (equal (farmer-side state) (wolf-side state))))
         nil)
        ((and (equal (goat-side state) (cabbage-side state))
              (not (equal (farmer-side state) (goat-side state))))
         nil)
        (t state)))
```

```
;;; +========+
;;; | Search |
;;; +========+

(defun path (state goal &optional (been-list nil))
  (cond
   ((null state) nil)
   ((equal state goal) (reverse (cons state been-list)))
   ((not (member state been-list :test #'equal))
    (or (path (farmer-takes-self state) goal (cons state been-list))
        (path (farmer-takes-wolf state) goal (cons state been-list))
        (path (farmer-takes-goat state) goal (cons state been-list))
        (path (farmer-takes-cabbage state) goal (cons state been-list)
    )))
```

```
;;; +==================+
;;; | Canned Execution |
;;; +==================+

(defun cross-the-river ()
  (let ((start (make-state 'e 'e 'e 'e))
        (goal (make-state 'w 'w 'w 'w)))
    (path start goal)))
```